1 Mechanics: Installing, Handing In, Demos, and Location of Documentation

1. To install, type `cs016_install convexhull` into a shell. The script will create the appropriate directories and deposit the stencil code files into them.

2. To hand in your project, go to the directory you wish to hand in, and type `cs016_handin convexhull` into a shell. Before using this script, make sure that you are in the correct directory! The script will hand in the entire current directory, and we don’t really want all of your files—just the ones pertinent to this project. (So be sure to save your README and any other files you need to hand in in the same directory as your code.)

3. To run a demo of this project, type `cs016_runDemo convexhull` into a shell.

4. To run the dummy version of the project, type `make run_dummy` into a shell.

5. To run the version of the project that uses your `MyHullFinder`, type `make run`.

6. For this project, you will be using the NSD4 Java library. Documentation is available at [http://net3.datastructures.net/doc4/index.html](http://net3.datastructures.net/doc4/index.html)

7. The Convex Hull documentation can be found here:
   This is also linked off of the class website on the Assignments page.

2 Introduction

2.1 What You’ll Do in this Project

In this assignment, you will implement a Graham Scan algorithm to find the convex hull of a set of points and to update the hull as necessary when new points are added to the set. You may optionally (for extra credit) also implement a Static Graham Scan, which computes the hull of a whole set of points at once, rather than updating the hull incrementally.

2.2 Purpose

The purpose of this assignment is to let you:
• Gain experience with computational geometry.
• Design and implement a complicated algorithm.
• Learn how annoying special and edge cases are, and how to deal with them.
• Understand the use of an auxiliary data structure in an algorithm.

### 2.3 The big picture

When your project’s complete, you’ll have a *visualizer* — an interface that lets you click on points, display the convex hull of the clicked points, clear the points, etc. It will look like this:

![Figure 1: The visualizer in action. Blue points are ones the user has clicked; red points are in the convex hull. In this example, the convex hull has been computed incorrectly!](image)

The visualizer has two parts: a graphical user interface, and an instance of a class that can do convex hull computations. The GUI, when you interact with it, makes calls to the convex-hull-finding class, which tells the GUI what is the current convex hull, so the GUI can draw it.

You’ll be writing a convex-hull-finding class, called *MyHullFinder*; we’ll also provide you another “convex-hull-finding” class, called *DummyHullFinder*, which gives the wrong answer when asked for the convex hull. It’s there so that when you start the project, you’ll have a program that runs and draws things, even if they’re the wrong things. Both of these classes implement the *ConvexHullFinder* interface, which we’ll discuss extensively below.

The graphical user interface portion of the project, which we’ve written for you, is called a *ConvexHullVisualizer*. And the part that assembles the two of them is the *ConvexHullProject* class. It consists of just a *main* method, which

• instantiates a *ConvexHullFinder*, and

• instantiates a *ConvexHullVisualizer*, passing it the finder to use.
The main method creates either a DummyHullFinder or a MyHullFinder, depending on whether you invoke it with an argument or not (any argument will do). To help you out, we’ve put all this in a Makefile: all you need to do is install the project and type

```
make run_dummy
```

or

```
make run
```

to invoke either the dummy version or your own version.

### 3 Overview of Your Tasks

We have provided stencil code for the MyHullFinder class, which you need to fill in. We recommend that you look at the DummyHullFinder class to get the general idea of how the hull finder should work, and that you run the project in dummy-finder mode (make run_dummy) to see how the methods of the hull finder are invoked by the user interface.
• You must implement the MyHullFinder class. This class must implement the interface support.convexhull.ConvexHullFinder, i.e., it’s a class to perform incremental and static convex-hull computations. It has only four methods:

– public void insertIncremental(HullPoint vertex)
– public void clear()
– public Iterator<Entry<Angle,HullPoint>> hull()
– public void angularGrahamScan(Vector<HullPoint> vertices)

and you need only implement the first three (the last one is for optional extra credit).

• Things to consider as you implement MyHullFinder:

1. A ConvexHullFinder must maintain the convex hull of a set of points. (Note that it need not maintain all the points – just their convex hull – although it may be useful for debugging to maintain the point set as well.)

2. When a ConvexHullFinder is created, the point set is empty, and the hull() method should, if invoked, return an empty list of points.

3. When insertIncremental is called, it represents the addition of a point to the point set. The ConvexHullFinder must compute the convex hull of the newly enlarged set. insertIncremental may be called multiple times before hull() is called again; nonetheless, it should update the ConvexHullFinder’s representation of the hull.

4. The point set may also be reset to the empty set (using clear()), or may be set all at once by invoking angularGrahamScan(Vector<HullPoint> vertices). When this method is called, the point set is replaced by the passed vertices, and the ConvexHullFinder should compute the convex hull in a single step.

5. The points in the convex hull will be stored in a CircularTree<Angle,HullPoint>, which is why the hull() method returns an iterator over Entry<Angle,HullPoint>s; we’ll explain this presently.

3.1 Requirement details

• You must implement an incremental Graham Scan algorithm. When a point is added to the set, your algorithm will determine whether it belongs in the convex hull, and if so, will then add it.

• You may optionally implement a static Graham scan as well, which should operate in time $O(n \log n)$, where $n$ is the number of points in the set passed to the static algorithm.
• You must implement the `ConvexHullFinder` exactly as it is defined; you cannot change the method signatures of anything in the stencil code, or the visualizer won’t work. You can, however, add as many helper methods as you want. We encourage this.

• The convex hull of a point set is, in our algorithm, defined by a collection of vertices. If three of the input points are collinear, they *could* all three be used as part of the hull, but we add the requirement that of any three collinear points, at most two of them may be on the convex hull returned by your `hull()` method.

• When you build your convex hull, you must store the set of points on the convex hull in $O(n)$ space, where $n$ is the number of points in the hull, not the total number of points on the screen. (You may also save the points from the original set in some other structure, if you want to use them for debugging, for instance.)

• Your `insertIncremental()` method, which adds a new point to the point set, and updates the convex hull, must run in $O(k \log n)$ time, where $k$ is the number of points removed from the hull to accommodate the new point, and $n$ is the number of points on the hull prior to the insertion.

• Your algorithm will compute angles from an anchor point; this anchor point must be a point *within* the convex hull. We recommend the following:
  – When there are only one or two points in the hull, use the first point as the anchor.
  – When there are more than two points, use the average of the first three points added to the hull as your anchor. Even if one of these points is later deleted from the hull, you need not update the anchor.
  – When the third point is added, and the anchor-point is updated, the angles for the first three points will all need to be recomputed, because they are measured relative to the anchor point.

### 3.2 Extra Credit: Static Graham Scan

Note that the Static Graham Scan method is extra credit: you are not required to implement it. The static Graham Scan takes in a set of points and finds the convex hull for that set all at once, rather than determining the convex hull as points are added.

### 3.3 README

The README for this project has some specific points that you need to address. Make a text file entitled “README,” which should be saved in the same directory as is your project; it will be handed in along with your code. (You can create a new text file by “opening” it in kate or any other editor—just `cd` into the directory in which you want to
save the README, then type, for example, \texttt{kate README \&} to create the file.) Each of the questions should be answered in about 2-4 lines, unless otherwise stated.

- Give an overview of your design for ConvexHull.
- Describe, briefly, any special cases which your insertion algorithm had to handle and how you chose to handle them.
- Additionally, please include descriptions of all of your test cases, as well as a brief explanation of why you believe that you have covered all possible cases. (This will be discussed further in Section 7) This section should be as long or as short as you find necessary.
- If you have anything else noteworthy about your program you haven’t mentioned yet, talk about it here (write as much or as little as you like).

4 Readings

- \texttt{Graham Scan Wikipedia Article}
- \texttt{Help Session Slides}
- \texttt{Spike’s Notes}

We’ll expect you to have read these materials before you start coding. Furthermore, the TAs may decline to help you until you’ve read them.

5 Using the Visualizer

We are providing you with a visualizer that will display a set of points and the convex hull you have computed for those points. You will be able to load pre-defined test cases of sets of points, and will also be able to create test cases directly in the visualizer, and save them to a file for later use. You can also print the set of points, and the set of hull points, to your console.

For the visualizer to work, you must correctly implement the Graham Scan algorithm, as well as the \texttt{hull()} method in the \texttt{MyHullFinder} class. (The visualizer won’t give you correct convex hulls if you do something wrong in the algorithm, but it won’t work at all if you forget to fill in \texttt{hull()}.)

When you run the visualizer, a window will appear with a column of buttons and a graphical background. Left click on the background to add points (make sure you don’t move the mouse during the click, or it won’t register the click). As you add points, the hull you’ve computed will be drawn with red lines, as long as the “incremental” radio-button at the bottom is selected. If “Static Graham Scan” is selected, then the hull is only displayed when you press the “Solve” button.

When a point is added (in incremental mode), it is classified and colored as follows:
• **Red:** The point is outside the current convex hull and gets added to the convex hull.

• **Blue:** The point is inside or on the current convex hull, so nothing changes.

• **Green:** Error: the point is invalid. This can be caused by internal java errors (e.g. `NullPointerException`), or by violating the requirements of this assignment. A fully functional project should not have any green points.

In static mode, the points are classified the same way, but only after the “Solve” button is pressed. Initially they are all drawn blue.

The visualizer has further features.

• You can add a point by specifying its x and y coordinates in the boxes on the upper left side of the window. This is especially useful for checking special cases.

• Right-clicking anywhere on the background will display the x and y coordinates of the click-point in the boxes on the right.

• If you click on “Clear,” it will remove all points currently in the drawing and allow you to start from scratch. (Be careful– this can’t be undone!)

• “Load File” allows you to input a pre-made test case to the visualizer. When you select “Load File,” a dialog box will appear, allowing you to select a file. To make your own files, you must adhere to a specific format. The file must consist of pairs of x and y pixel coordinates representing the points \(^1\) formatted as in the following example:

\[
\begin{align*}
100 & 100 \\
100 & 50 \\
234 & 453 \\
74 & 222 \\
\end{align*}
\]

Loading this file will create 4 points, with the coordinates (100, 100), (100, 50) . . . . There must be only one pair of numbers per line, and there must be one pair of numbers on every line. (That is, no blank lines are allowed in the file.) Creating files is **highly encouraged**, as it makes repeat testing of special cases very simple.

6 Your Code

6.1 Design

6.1.1 Data Structures

The data structure you must use to store the hull is the `CircularTree` that we provide for you in the `support.convexhull` package.

---

\(^1\)The point (0, 0) is in the upper left corner of the canvas, and the y coordinate increases as you move down, while x increases as you move right. Limit your values to lie between 0 and 500, or the points may appear offscreen, which could confuse you.
Note that, unlike in the Heap project, you are not designing a generic data structure for Convex Hull. Rather, you are instantiating the CircularTree class that is itself generic, so it is your responsibility to fill in the concrete classes that the tree will use for its key K and value V. For the visualizer to properly display your hull, it is required that the key be of type Angle and the value or element stored of type HullPoint.

6.2 Angles

You will be storing points in the CircularTree using the Angle between the new point and the anchor point as the key. These Angles will keep them organized in counterclockwise order. (See the readings for further clarification.) As we said above, initially, the anchor point will be the first point you add to the hull. This will change once you have three non-collinear points in the hull: at that point, a new anchor point should be calculated by finding the midpoint of the three points currently in the hull. You will then need to update the Angles of the points already in the hull and reinsert them into the tree based on these new keys.

To make a new Angle, you will instantiate a new instance of the Angle class, passing it the difference of the x-coordinates and the difference of the y-coordinates between the new point and the anchor point. (See the readings for more information.)

For thinking through test-cases, remember that the upper left-hand corner of the canvas is (0, 0); this means that points which are lower on the screen will have a higher y-value than the points above them. Note that this is unlike the standard coordinate system upon which the Angle class is based. (Points on the screen follow the same x-coordinate convention as in the Angle class.) That means that a polygon that’s counterclockwise in ordinary coordinates is clockwise in screen coordinates, and vice versa. We recommend that you do all your work in ordinary Cartesian coordinates, so that your convex hull, if drawn one-segment-at-a-time, would show up in clockwise order on the screen.

6.3 Orientation Test

As described above, when you add a new point to your CircularTree, it will be ordered according to the Angle calculated between itself and the anchor point. Using this ordering, you must be able to judge whether three ordered points are oriented clockwise or counterclockwise. (You can also see the readings for more help on this topic.) When checking the orientation of three points, you should return an int—this will allow you to use the built-in <, >, ==, etc. operators. For example, you could return −1 to indicate counterclockwise, 1 to indicate clockwise, and 0 to indicate collinear points.

The class support.convexhull.HullPoint maintains coordinate information about the points. A HullPoint is created and passed into your insert method every time a point is added to the canvas of the visualizer. Your insert method must determine whether this point should be added to the hull. The HullPoint class provides the coordinates that your orientation tester should be using when performing tests on points.
7 Testing

For this project, you will need to hand in test cases. We’re not requiring a specific number of test cases, but rather a well thought-out set of cases which demonstrate that your project is fully functional, no matter what input it is run on.

To create your test cases, you may either directly write them in a file, or you may use the visualizer to “draw” a test case, then save it to a file. (Both of these methods are described in Section 5.) In your README, you should also give a clear explanation of exactly what each test is checking, as well as a brief explanation of why you believe that you have covered all possible test cases. Make sure that it is easy for your TAs (i.e., your graders) to match a test file with its explanation in the README!

8 What to Hand In

1. Code for the class MyHullFinder.

2. A README pointing out any bugs or other problems in your code (or an assertion that there aren’t any), as well as describing any significant design choices. For this README, make sure to also include everything discussed in Section 3.3 and in Section 7.

3. A set of test cases which demonstrate the full functionality of your project. (Don’t forget to include descriptions of your test cases in the README, as described in Section 7.)